Targeting heterogeneous architectures via macro data flow

M. Aldinucci†, M. Danelutto°, P. Kilpatrick⋆, M. Torquati°

Univ. of Torino†
Ph: +39 0116706852
Fax: +39 0116706711
aldinuc@di.unito.it

Univ. of Pisa°
Ph. +39.050.2212742
Fax: +39.050.2212726
{marcod, torquati}@di.unipi.it

Queen's Univ. of Belfast⋆
Ph: +44 (0)28 9097 4648
Fax: +44 (0)28 9097 4879
p.kilpatrick@qub.ac.uk

***Abstract*—We propose a data flow based run time system as an efficient tool for supporting execution of parallel code on heterogeneous architectures hosting both multicore CPUs and GPUs. We discuss how the proposed run time system may be the target of both structured parallel applications developed using algorithmic skeletons/parallel design patterns and also more "domain specific" programming models. Experimental results demonstrating the feasibility of our approach are presented.**

Keywords: data flow, structured parallelism, algorithmic skeletons, parallel design patterns, heterogeneous architectures.

I. Introduction

Several technological developments have directed computer manufacturers–previously seeking ever more powerful single core CPUs–toward the development of increasing numbers of parallel processing elements. As a consequence, heterogeneous systems composed of one or more processing nodes, each hosting one or more multicore CPUs and one or more GPUs are becoming a *de facto* standard in a wide range of systems, from mobile phones and tablets to workstations, and on to high end parallel computer systems.

Such heterogeneous systems raise new challenges related to the programming models. While multicore programming may make better use of multi-threading technology and GPUs may be mastered with relatively low level tools such as OpenCL or equivalent proprietary frameworks such as Nvidia CUDA, no uniform, high level programming model exists suitable for exploiting both components of these common heterogeneous systems. As a result, programmers writing applications for heterogeneous systems must be able to master radically different programming techniques to develop efficient parallel computations on both parts of these systems: CPUs and GPUs.

The main reason for the lack of uniformity lies in the essentially different nature of the two components of these systems: CPUs are general purpose devices, suitable for supporting with similar efficiencies both data parallel and stream/control parallel computations, while GPUs only provide efficient support for data parallel computations. Also, while relatively high level parallel programming models and tools exist for CPUs, the programming models provided by GPU vendors and those developed for platform independent GPU general purpose computing notably present a lower level of abstraction. In particular, memory allocation and management is completely the responsibility of the GPU application programmer and quite profound knowledge of the hardware features of the GPU (e.g. number of cores per controller, dimensions of the different kinds of memory present, etc.) is needed in order to design and implement *efficient* GPU programs.

In this work, we propose a different approach to the development of efficient applications for heterogeneous systems. The approach is hierarchical. At the lowest level, we propose to virtualize the peculiarities of the different heterogeneous components by developing suitable "task executor" run times on both GPUs and CPUs. While CPU executors support efficient execution of any kind of task, including sequential tasks, the GPU executors will support only efficient execution of *data parallel* tasks. At the higher level, we propose ways of compiling design pattern [17] (algorithmic skeleton [12]) based parallel applications as well as more domain specific kinds of application–such as those typical of the numerical mathematical community–to macro data flow graphs. These graphs are subsequently processed by a macro data flow interpreter scheduling tasks to the run time executors running on the available cores and GPUs.

This two step process presents several advantages: i) it raises the level of abstraction presented to the application programmer by the programming framework ii) it provides suitable points at which to implement well-known optimization techniques both at compile time (macro data flow graph compiler) and at run time (macro data flow interpreter/task scheduler) iii) it seamlessly integrates CPUs and GPUs within the same programming model; and iv) it relieves the application programmer of the necessity to deal with hardware related features when programming GPUs.

The rest of the paper is structured as follows: Sec. II introduces the macro data flow model. Sec. III and IV explain how macro data flow is compiled from high level languages and eventually efficiently executed on heterogeneous architectures. Sec. V discusses the results of our experiments aimed at validating the proposed approach. Sec. VI discusses related work and Sec. VII concludes the paper.

II. Macro data flow

Data flow is the well-know computing [15] model where the "control flow" of a program is determined only by the data

[0]This work has been partially supported by the EU STREP ParaPhrase

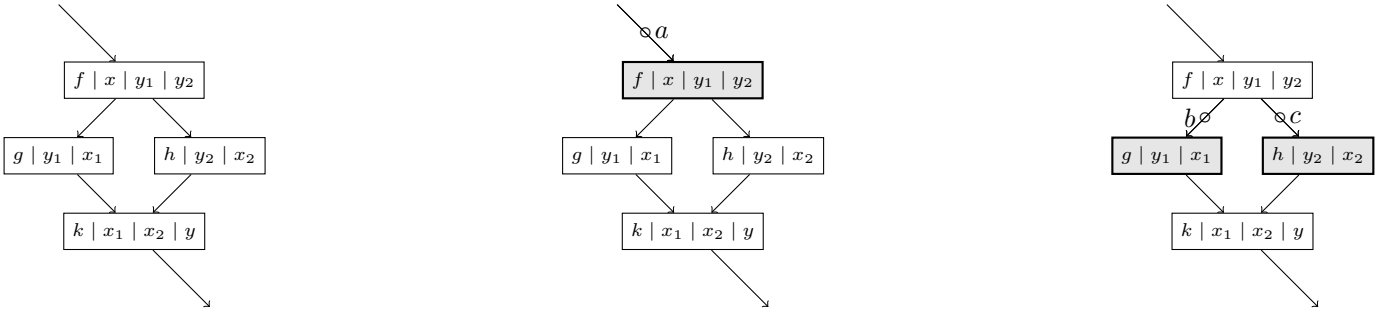

Fig. 1. Macro Data Flow program/graph (left) and snapshots of its execution (right, fireable instructions outlined)

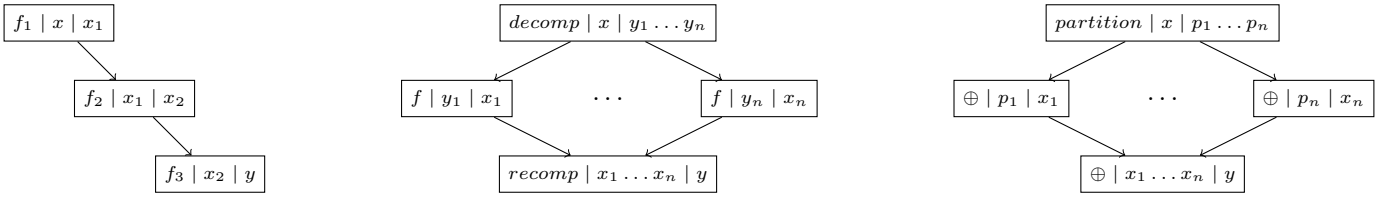

Fig. 2. Compiling skeletons to MDF: $pipe(f_1, f_2, f_3)$ (left), $map(f)$ (mid) and $reduce(\oplus)$ (right)

precedences among functions transforming input into output. In the past, *ad hoc* architectures have been designed aimed at implementing in hardware the data flow model [9], [13]. Mainly as a result of the typical grain of computation executed in parallel on these architectures (basically ILP) and of the technology used, they never succeeded in replacing classical Von Neumann architectures.

"Macro" data flow (MDF, [8], [6], [3], [14]) builds on the plain data flow model by augmenting the grain of the computations scheduled for execution according to their data dependencies. Full portions of sequential code are considered as macro data flow "instructions", provided they implement pure functions mapping input to output data. A MDF program is therefore represented as a MDF graph whose nodes represent functions implemented by means of sequential portions of code wrapped in functions/procedures/methods while arcs represent data flow dependencies.

The execution of a MDF program starts with the assignment of the input data ("tokens" in data flow jargon) to the input arcs of the first instruction in the MDF graph and proceeds with a loop. The loop body looks for "fireable" instructions in the MDF graph–instructions with all the input data available (all input arcs with a token present)–and executes these instructions, possibly in parallel. The results of the fireable instructions executed are then directed to the destination MDF instructions identified by the output arcs of the instruction, and the loop is restarted. The program terminates when there are no more fireable instructions or data tokens on arcs directed to other instructions. Fig. 1 shows a typical MDF graph (left) and two different configurations of the same graph during its execution.

Stream parallelism and data parallelism are modeled using MDF in two different ways. *Stream parallelism* is managed by creating a new "fresh" copy of the MDF graph derived from the compilation of the program onto the single stream item for each item appearing on the input stream and passing that item as the input token of the graph. Therefore stream parallelism derives from the execution of fireable instructions from different graph instances. This requires labeling of the graph instructions with an additional graph tag, of course. *Data parallelism* is managed by inserting into the graph instructions which "decompose" their input data structure(s) into multiple data sets and direct these data sets to a number of independent instructions computing partial results which are eventually directed to another instruction "rebuilding" the final result out of the multitude of partial results (see Fig. 2).

III. Compiling HLL to MDF

We consider two classes of high level programming frameworks: one based on parallel design patterns/algorithmic skeletons and another based on high level numerical algorithms.

a) Algorithmic skeletons: We consider a quite classic skeleton framework providing stream parallel skeletons (pipeline, farm) and data parallel skeletons (map, reduce, stencil) supporting a two-tier nesting model such as that introduced by P3L [] and adopted by Muesli []: data parallel skeletons should have either data parallel or sequential nested skeletons; stream parallel skeletons may have stream parallel, data parallel or sequential nested skeletons. Fig. 2 shows the compilation schemes for pipeline and data parallel skeletons. Farm is eliminated (substituted by stream parallelism implemented with multiple instances of the MDF graph). Nesting is naturally handled by composing MDF graphs under the assumption that graphs deriving from compilation of a single

```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k] := DSYRK(A[k][n], A[k][k])
  A[k][k] := DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k] :=
        DGEMM(A[k][n], A[m][n], A[m][k])
    A[m][k] := DTRSM(A[k][k], A[m][k])
```

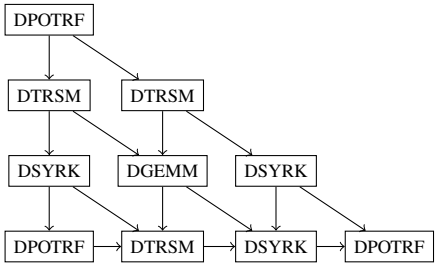

Fig. 3. Cholesky factorization: high level code and MDF graph (3×3 tiles)

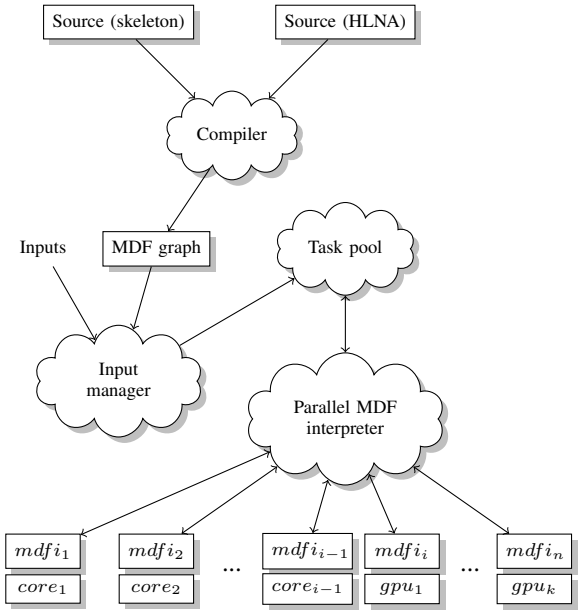

Fig. 4. HLL to MDF workflow

skeleton always have a unique input arc and output arc. Referring to Fig. 2, nesting is implemented by allowing the different pipeline stage instructions or map/reduce workers to be substituted by entire MDF sub-graphs deriving from the compilation of the inner parameter skeletons.

b) High level numerical algorithms: We consider applications computing high level numerical algorithms (HLNA) expressed as sequences and/or nestings of loops with calls to functions of numerical libraries (e.g. BLAS)[1]. In this case the MDF graph is derived by using the same algorithms used to implement data flow analysis in sequential compilers. The kind of graphs derived from high level numerical algorithms are those discussed in [16]. Fig. 3 shows the graph derived from classic Cholesky factorization.
A more detailed description of how MDF graphs are generated from HLL and/or libraries may be found in [1].

IV. Targeting heterogeneous architectures

In order to target heterogeneous architectures, we designed a distributed MDF interpreter as follows (see Fig. 4):

- a logically centralized *task pool* hosts all the instances of the MDF graph submitted for execution. One instance of the compiled MDF graph is inserted in the task pool by the *Input Manager* upon the availability of an input data set (input token). The task pool is logically centralized. However, on highly parallel machines, it will be implemented in a distributed way to avoid bottlenecks. We have preliminary experiments demonstrating the feasibility of the implementation of the task pool as a distributed task pool tree, with leaves distributed on the nodes of the NUMA architecture serving the local cores/GP-GPUs and inner nodes contributing to ensure load balancing (e.g. implementing fireable instructions stealing). The preliminary results show that the performance penalty is negligible and that removal of the bootleneck implied by the logical centralized task pool may be achieved.
- each (GP-)core in the architecture runs an *interpreter instance* ($mdfi$) performing as the interpreter loop described in Sec. II: fetch a fireable MDF instruction, execute, store back results.
- for each GP-GPU, a further thread is run, performing a slightly different interpreter loop: the loop begins by looking for *data parallel only* fireable instructions in the task pool, and then continues by off-loading to the associated GPU the execution of these data parallel MDF instructions. All the details relative to data movement to and from GPU as well as to memory allocation and to thread scheduling on the GPU are managed by this modified interpreter loop. It is worth pointing out that the thread managing the GP-GPU has full control over the computations (and associated data) scheduled to the GP-GPU. Therefore, efficient caching policies may be implemented to avoid unnecessary traffic on the PCIe bus moving data to and from GPU memory. In particular, all those data potentially reused during different phases (MDF instruction execution) of the computation may be kept on the GP-GPU memory up to the point when memory is needed for other data structures. If the thread managing the GP-GPU realizes that (part of) the data needed to schedule the next data parallel MDF instruction to the GP-GPU are already on the GP-GPU memory, it will simply schedule the MDF data parallel instruction without re-sending data through the PCIe bus.

[1] we assume to have programs written in C/C++ code in this case, although our methodology does not actually rely upon a particular sequential host language

The key point here is the scheduling of "data parallel only" instructions to the GPU nodes. We consider two kinds of data parallel instruction: i) single MDF instructions for which a GPU implementation exists (this is the case for BLAS routines, for example), or ii) MDF subgraphs deriving from the compilation of map-like patterns (map, reduce, stencil, ...), i.e. those subgraphs made of a decompose instruction, a recompose instruction and of n "worker" instructions. In the latter case we implement the full decomp/compute/recomp subgraph on the GPU–possibly scheduling computation of different partitions of workers in kernels of a GPU stream, thus overlapping (part of) the data transfer cost.

Compile options are provided to choose between CPU-only, GPU-only or mixed CPU-GPU execution of data parallel MDF sub-graphs. The code used to feed GP-GPUs is ideally derived from the high level source code provided by the user without actually requiring the application programmer to write any specific GP-GPU (e.g. CUDA or OpenCL) code. This is possible because the high level code exposes all the functional parameters needed to generate efficient GP-GPU code at compile time. Furthermore, we are currently refining several autonomic strategies to devise on-the-fly the most suitable execution strategy (CPU, GPU or mixed) without programmer intervention. These strategies schedule different partitions of the data parallel sub-graphs on both CPU and GPU, monitor execution times and eventually decide where subsequent partitions are scheduled for execution. Monitoring continues during the full data parallel computation with a minimal overhead to ensure decisions taken in the initial phase are verified and possibly changed if need be.

V. Experiments

We performed some preliminary experiments to validate the effectiveness of the MDF approach to target heterogeneous architectures. The experiments have been run on a single heterogeneous machine with two AMD *Magny cours* Opteron 12 core CPUs and a single C2050 Nvidia GPU. In all the experiments we used a synthetic skeleton application whose parallel structure is basically a map skeleton, that is with the same parallel structure as that appearing in a matrix multiplication algorithm, when using the naive algorithm

$$forall\ i\ do\ C_i = A_i \times B$$

First we verified that when enabled to target both CPU and GPU, the MDF interpreter succeeds in exploiting both kinds of core. Fig. 5 shows the average number of tasks computed by CPU core(s) and by the GPU. The "non-optimized" version implements the naive algorithm: for each MDF instruction $\times(A_i, B)$, parameters are copied to and from the GPU. That is, each $\times(A_i, B)$ instruction involves the copying of both row A_i and matrix B to GPU and of the result C_i row from the GPU. The "optimized" version, recognizes that B is a common parameter and thus implements the execution of *all* the instructions $\times(A_i, B)$ as follows: the first instruction copies B to GPU memory. All the others assume that B is already on the GPU and therefore only copy A_i to and C_i from the GPU. As expected, the number of tasks executed on the GPU become smaller and smaller as more and more CPU cores are used. Initially, the GPU executes about 90% of the tasks, but when a larger number of cores is used, a smaller percentage of tasks are executed on the GPU. Moreover, when GPU copies are optimized, and thus the GPU execution of the MDF instructions becomes faster than the execution of the same instructions on the CPU cores, the GPU succeeds in executing more tasks.

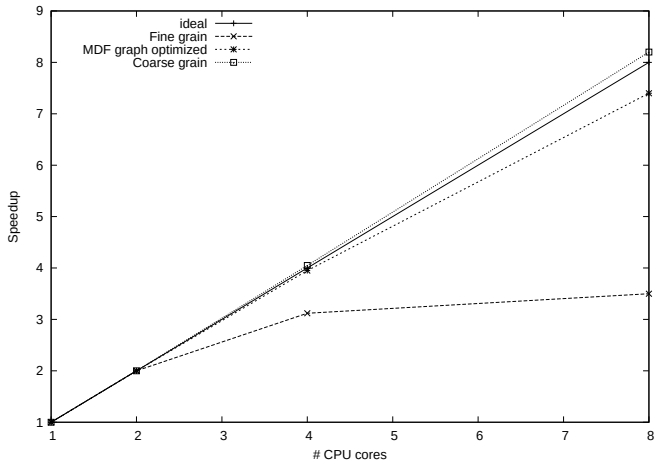

Fig. 6. Grouping map "worker" MDF instructions on CPU cores

It is worth pointing out that, also in the case when only CPU cores are used, other strategies applying similar kinds of optimization may be implemented. Fig. 6 shows the completion times and speedups achieved when executing a $map(map(f))$ application on CPU cores only. The "Fine grain" execution, refers to the plain execution of all the inner map "worker" MDF instructions as they are generated by the compiler. The "Coarse grain" execution, instead, refers to execution of the $map(seqmap(f))$ version of the same application, obtained by applying the skeleton rewriting rule

$$map(map(f)) \equiv map(seq(\forall\ i\ (f)))$$

The "MDF graph optimized" execution refers to the execution of the original $map(map(f))$ application where the derived macro data flow graph is optimized "on-the-fly" to group the worker MDF instructions relative to the same map skeleton. This optimization is completely general, may be applied in all cases where a map skeleton with a (too) large number of worker MDF instructions is present in the compiled MDF graph, and achieves performances comparable to those achieved by the source-rewritten application code.

Finally, Fig. 7 shows the differences in the completion times measured with GPU optimized vs. non-optimized execution of the same synthetic map application. In this case, the optimization of the unnecessary B matrix copies introduces an improvement in the completion time in the $[20\%, 30\%]$ range.

#cores	*unoptimized GPU copies*			*optimized GPU copies*		
	Avg task/core	GPU tasks	GPU %	Avg task/core	GPU tasks	GPU %
1	53	459	89%	43	469	91%
2	53	406	79%	37	438	85%
4	49	317	61%	34	377	73%
8	40	193	37%	30	272	53%
16	30	42	8%	23	141	27%

Fig. 5. CPU/GPU task distribution

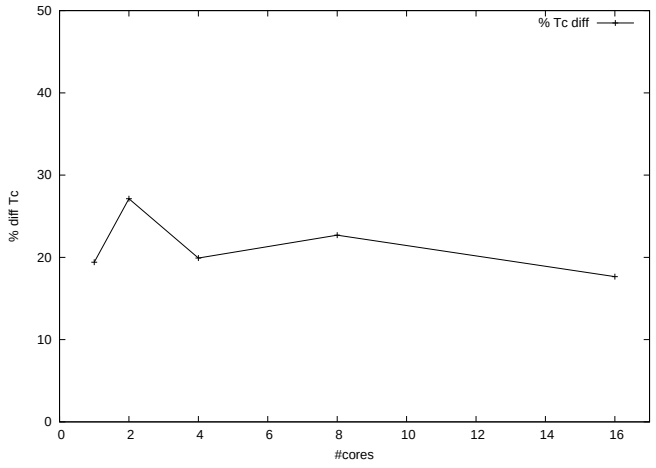

Fig. 7. Percentage difference in completion times of the execution of the application with and without copy optimization

VI. Related work

Heterogeneous architectures hosting multicores and GPUs are currently programmed using a mix of multi-threading and CUDA/OpenCL techniques. In this case, all the burden of heterogeneous device exploitation is the responsibility of the application programmer. Researchers active in the algorithmic skeleton area have designed and implemented skeleton frameworks exploiting CPU *and* GP-GPUs in the execution of skeleton code. Notable examples are the SkePU framework [10], Muesli [11] and SkeCL [19]. Despite the fact that some of these frameworks appear more mature than ours (they can already be downloaded from the group web pages), none of them uses macro data flow, and most of the compilation techniques used are static. In addition, while SkePU and Muesli target both components of heterogeneous architectures (i.e. CPU and GP-GPU cores), SkeCL only uses GP-GPU cores.

Programming frameworks based on algorithmic skeletons have been recently introduced to alleviate the task of the application programmer when targeting data parallel computations to GPUs. Muesli [11] supports GPU targeting for data parallel computations under programmer responsibility (the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons). SkePU [7] provides programmers with GPU implementations of map and reduce skeletons and relies (in the latest versions) on StarPU for the execution of stream parallel skeletons (pipe and farm). Both environments adopt a template based approach rather than a macro data flow based implementation for skeletons.

The OpenMP task concept [4] has also been extended to deal with data dependencies in StarSS [5]. This led to the possibility to define *de facto* MDF graphs by suitably annotating sections of code in the source C/C++ application. The main difference with our approach consists in the "declarative" (annotation style) approach followed which requires considerable application programmer intervention, as the programmer must fully understand the business logic of the application in order to write effective and efficient annotations. A number of different projects aim at extending the "task" concept to support automatic and efficient scheduling of tasks on multicore heterogeneous machines [18], [16]. However, the responsibility for identifying the tasks and the related dependencies is left to the application programmer. On the one hand, this allows a higher degree of freedom in the definition of the parallel computation for the application programmer. On the other hand, the level of abstraction presented to the applications programmer by the programming framework is much lower than in our approach.

VII. Conclusions

We briefly introduced a two phase process targeting heterogeneous architectures built of multi-cores and GPUs. The first step is aimed at translating high level languages into macro data flow graphs. These graphs are then executed by means of a parallel macro data flow interpreter specialized to run data parallel computations on GPUs without programmer intervention. The preliminary experimental results show that the approach is feasible and efficiently implements different kinds of applications on a heterogeneous, single node architecture.

A number of different optimizations may be used when targeting heterogenous architectures by MDF, making the best (and correct) usage of the information coming either from the high level skeletons or from the structure of the MDF graphs. We are currently investigating the possibilities offered by several techniques, including the possibility to monitor execution of data parallel instructions on CPU cores/GPUs and to autonomically decide to schedule them on either GPU or CPU cores using Behavioural skeleton technology [2] and the possibility to use affinity scheduling and job stealing techniques to improve the memory usage in NUMA heterogeneous architectures.

REFERENCES

[1] M. Aldinucci, L. Anardu, M. Danelutto, P. Kilpatrick, and M. Torquati. Parallel patterns + Macro Data Flow for multi-core programming. In *Proceedings of EuroMicro PDP*. IEEE press, 2012. accepted, to appear.

[2] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Towards hierarchical management of autonomic components: a case study. In Didier El Baz, Tom Gross, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing*, pages 3–10, Weimar, Germany, February 2009. IEEE.

[3] Samer Arandi and Paraskevas Evripidou. Programming multi-core architectures using Data-Flow techniques. In *ICSAMOS'10*, pages 152–161, 2010.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

[5] R. Badia. StarSs support for programming heterogeneous platforms. In *Proceedings of Heteropar 2012 workshop, to appear in EuroPar 2012 Parallel Computing Workshop Proceedings*, 2012.

[6] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36:29–35, June 2009.

[7] U. Dagstgeer, C. Kessler, and S. Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *Proceedings of PARCO 2012*, Gent, 2012. to appear, Gent.

[8] Marco Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[9] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[10] Johan Enmyren and Christoph W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[11] S. Ernsting and H. Kuchen. Data Parallel Skeletons for GPU Clusters and Multi-GPU Systems. In *Proceedings of PARCO 2012*, Gent, 2012. to appear.

[12] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw., Pract. Exper.*, 40(12):1135–1160, 2010.

[13] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[14] British Jetley and Laxmikant V. Kal. Static macro data flow: Compiling global control into local control. In *IPDPS Workshops'10*, pages 1–8, 2010.

[15] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[16] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22:15–44, 2010.

[17] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[18] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with starss. *IJHPCA*, 23(3):284–299, 2009.

[19] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA, 2011. IEEE Computer Society.